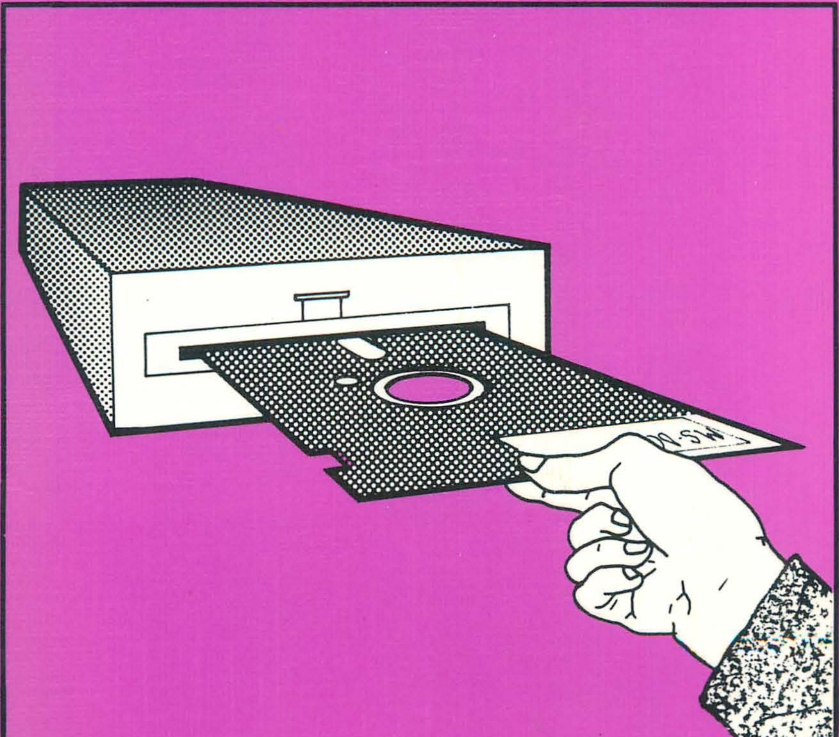
COVERS ALL VERSIONS OF 2.x, 3.x AND 4.x

A Concise Advanced User's Guide to MS-DOS

N. KANTARIS



A Concise Advanced User's Guide to MS-DOS

A Concise Advanced User's Guide to MS-DOS

by
Noel Kantaris

BERNARD BABANI (publishing) LTD.
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

First Published – July 1989
Reprinted – March 1990

British Library Cataloguing in Publication Data:
Kantaris, Noel
A concise advanced user's guide to MS-DOS
1. Micro-computer systems. Operating systems. MS-DOS
I. Title
005.4'469

ISBN 0 85934 209 3

Typeset direct from disc by Commercial Colour Press, London E7.
Printed and Bound in Great Britain by Cox & Wyman Ltd, Reading

ABOUT THE BOOK

This Concise Guide to Advanced User Guide to MS-DOS has been written for those who already have some knowledge of MC/PC-DOS commands, but who would like to be able to write customised batch files, create specialist programs with the use of the **debug** program and, in general, extend their abilities towards designing and setting up their own professional looking menu screens so that they or others could run any program application or package easily.

The book was not designed to teach you how to use DOS commands in a routine manner. If you need to know about this aspect of DOS, then may I suggest that you refer to another book, entitled *A Concise Introduction to MS-DOS (BP232)*, also published by the Bernard Babani (publishing) Ltd.

This concise guide was written with the busy person in mind. You don't need to read hundreds of pages to find out most there is to know about a subject, when a few pages can do the same thing quite adequately! With the help of this book, it is hoped that you will be able to get the most out of your computer in terms of efficiency and productivity, and that you will be able to do it in the shortest, most effective and informative way.

ABOUT THE AUTHOR

Graduated in Electrical Engineering at Bristol University and after spending three years in the Electronics Industry in London, took up a Tutorship in Physics at the University of Queensland. Research interests in Ionospheric Physics, lead to the degrees of M.E. in Electronics and Ph.D. in Physics. On return to the UK, he took up a Post-Doctoral Research Fellowship in Radio Physics at the University of Leicester, and in 1973 a Senior Lectureship in Engineering at The Camborne School of Mines, Cornwall, where since 1978 he has also assumed the responsibility of Head of Computing.

TRADEMARKS

ACKNOWLEDGEMENTS

I would like to thank colleagues at the Camborne School of Mines for the helpful tips and suggestions which assisted me in the writing of this book.

CONTENTS

INTRODUCTION

Most commercial software are designed with 'user-friendly' screens incorporating such screen attributes as reverse video and colour, with information appearing in the right place on the screen. MS-DOS can be utilized to do just that, provided you know how to do it. To this end, you will be shown how to write specialized batch files with the use of the **edlin** line editor, and how to design your own screen menus. You could, of course, buy a commercially available program that could do all this, but then it would cost you a lot and you would not learn anything new.

You might already have written batch files to allow you to run easily an application, but creating a professionally looking batch file requires you to write some specialised programs in assembler. To this end, you will be shown how to use the **debug** program to write programs which control the appearance of the cursor, without necessarily having to become an expert assembler programmer. In general, you will be shown how to extend your abilities towards designing and setting up your own professional looking menu screens so that you or others could choose and run program applications or packages easily.

Although the internal DOS commands provide control over the disc drives and, to some lesser extend, control over the keyboard and display screen, the appearance of the latter can be controlled far more effectively with the ANSI.SYS driver, which is an external program supplied with your MS/PC-DOS operating system. Every device that is connected to your computer is controlled by such an external program, usually having the filename extension SYS. However, before any ANSI.SYS command could be used, you must make sure that the path is accessible from the root directory of your system's disc and that the extra line DEVICE=ANSI.SYS is included in the CONFIG.SYS file.

If you are not absolutely sure what is meant by the contents of the last paragraph then refer first to Appendix A, entitled 'System Configuration', which was included here for completeness, but might not be of great value to the more experienced DOS user. However, if things are still not clear after you have referred to this section, then perhaps you should refer to the book *A Concise Introduction to MS-DOS*, also published by Bernard Babani (BP 232), as it might be more appropriate for you at this stage with its much lower entry point into DOS.

The ASCII Code of Character Conversion:
The ASCII code (which stands for American Standard Code of Information Interchange) is the accepted standard for representing characters in computers. It defines codes 0 to 127; the first 32 (codes 0 to 31) as control characters, which define some action such as line-feed or form-feed, and the remaining (codes 32 to 127) as standard characters which normally appear on a computer keyboard. Since each byte can represent one of 256 possible characters, there are another 128 codes available (codes 128 to 255) for which, however, there is no formal standard laid down. These codes are used by IBM and IBM compatibles and are known as the IBM extended character set.

The IBM extended character set includes four main groups:

(a) Accented international characters (codes 128 to 168);
(b) Line drawing characters (codes 169 to 223);
(c) Greek letters (codes 224 to 239), and
(d) Mathematical symbols (codes 240 to 254).

All the codes are shown in the ASCII Conversion Codes table which appears in the next two pages. The table shows all 256 characters and tabulates their values in both decimal (base 10) and hexadecimal (base 16) representation. All, but one, ASCII codes can be entered into the computer by holding the key marked **Alt** down and typing the decimal character code on the numeric keypad (not the numbers appearing on the first row of keys of the normal keyboard. On releasing the **Alt** key, the corresponding character appears on the screen. Thus, typing

```
C:\> Alt-236
```

causes the symbol for infinity (∞) to appear on the screen. The one character code that can not be entered with this method is the 'null' character (code 0).

The first 32 character codes (0 to 31) can also be entered with the **Ctrl** key, as indicated in the ASCII Conversion Codes table. Using this method, however, causes DOS to echo the caret (^) character followed by the corresponding letter on the screen. **Edlin,** like DOS, allows you to enter the control characters with either the **Ctrl** key or the **Alt** key, but always echoes a caret followed by the appropriate letter. To enter character 0 into a file, press F7 while using **edlin** or the **debug** program (the latter one of which will be discussed in some detail later).

TABLE 1 ASCII Conversion Codes

CHAR	ABBR	DEC	HEX	CHAR	ABBR	DEC	HEX	CHAR	ABBR	DEC	HEX
CTRL @	nul	0	00	CTRL K	vt	11	0B	CTRL V	syn	22	16
CTRL A	soh	1	01	CTRL L	ff	12	0C	CTRL W	etb	23	17
CTRL B	stx	2	02	CTRL M	cr	13	0D	CTRL X	can	24	18
CTRL C	etx	3	03	CTRL N	so	14	0E	CTRL Y	em	25	19
CTRL D	eot	4	04	CTRL O	si	15	0F	CTRL Z	sub	26	1A
CTRL E	enq	5	05	CTRL P	dle	16	10	CTRL [	esc	27	1B
CTRL F	ack	6	06	CTRL Q	dc1	17	11	CTRL \	fs	28	1C
CTRL G	bel	7	07	CTRL R	dc2	18	12	CTRL]	gs	29	1D
CTRL H	bs	8	08	CTRL S	dc3	19	13	CTRL ^	rs	30	1E
CTRL I	ht	9	09	CTRL T	dc4	20	14	CTRL	us	31	1F
CTRL J	lf	10	0A	CTRL U	nak	21	15				
SPACE		32	20	@		64	40			96	60
!		33	21	A		65	41	a		97	61
"		34	22	B		66	42	b		98	62
#		35	23	C		67	43	c		99	63
$		36	24	D		68	44	d		100	64
%		37	25	E		69	45	e		101	65
&		38	26	F		70	46	f		102	66
'		39	27	G		71	47	g		103	67
(		40	28	H		72	48	h		104	68
)		41	29	I		73	49	i		105	69
*		42	2A	J		74	4A	j		106	6A
+		43	2B	K		75	4B	k		107	6B
,		44	2C	L		76	4C	l		108	6C
-		45	2D	M		77	4D	m		109	6D
.		46	2E	N		78	4E	n		110	6E
/		47	2F	O		79	4F	o		111	6F
0		48	30	P		80	50	p		112	70
1		49	31	Q		81	51	q		113	71
2		50	32	R		82	52	r		114	72
3		51	33	S		83	53	s		115	73
4		52	34	T		84	54	t		116	74
5		53	35	U		85	55	u		117	75
6		54	36	V		86	56	v		118	76
7		55	37	W		87	57	w		119	77
8		56	38	X		88	58	x		120	78
9		57	39	Y		89	59	y		121	79
:		58	3A	Z		90	5A	z		122	7A
;		59	3B	[		91	5B	{		123	7B
<		60	3C	\		92	5C	|		124	7C
=		61	3D	]		93	5D	}		125	7D
>		62	3E	^		94	5E	~		126	7E
?		63	3F	_		95	5F	del		127	7F

TABLE 1 (Contd.)

CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX
Ç	128	80	½	171	AB	╓	214	D6
ü	129	81	¼	172	AC	╫	215	D7
é	130	82	¡	173	AD	╪	216	D8
â	131	83	«	174	AE	┘	217	D9
ä	132	84	»	175	AF	┌	218	DA
à	133	85	░	176	B0	█	219	DB
å	134	86	▒	177	B1	▄	220	DC
ç	135	87	▓	178	B2	▌	221	DD
ê	136	88	│	179	B3	▐	222	DE
ë	137	89	┤	180	B4	▀	223	DF
è	138	8A	╡	181	B5	α	224	E0
ï	139	8B	╢	182	B6	ß	225	E1
î	140	8C	╖	183	B7	Γ	226	E2
ì	141	8D	╕	184	B8	π	227	E3
Ä	142	8E	╣	185	B9	Σ	228	E4
Å	143	8F	║	186	BA	σ	229	E5
É	144	90	╗	187	BB	µ	230	E6
æ	145	91	╝	188	BC	τ	231	E7
Æ	146	92	╜	189	BD	Φ	232	E8
ô	147	93	╛	190	BE	Θ	233	E9
ö	148	94	┐	191	BF	Ω	234	EA
ò	149	95	└	192	C0	δ	235	EB
û	150	96	┴	193	C1	∞	236	EC
ù	151	97	┬	194	C2	φ	237	ED
ÿ	152	98	├	195	C3	ε	238	EE
Ö	153	99	─	196	C4	∩	239	EF
Ü	154	9A	┼	197	C5	≡	240	F0
¢	155	9B	╞	198	C6	±	241	F1
£	156	9C	╟	199	C7	≥	242	F2
¥	157	9D	╚	200	C8	≤	243	F3
₧	158	9E	╔	201	C9	⌠	244	F4
ƒ	159	9F	╩	202	CA	⌡	245	F5
á	160	A0	╦	203	CB	÷	246	F6
í	161	A1	╠	204	CC	≈	247	F7
ó	162	A2	═	205	CD	°	248	F8
ú	163	A3	╬	206	CE	∙	249	F9
ñ	164	A4	╧	207	CF	·	250	FA
Ñ	165	A5	╨	208	D0	√	251	FB
ª	166	A6	╤	209	D1	ⁿ	252	FC
º	167	A7	╥	210	D2	²	253	FD
¿	168	A8	╙	211	D3	■	254	FE
⌐	169	A9	╘	212	D4		255	FF
¬	170	AA	╒	213	D5			

Computers work with the *binary* representation (base 2), but as this is too difficult to translate, we tend to work in decimal or hexadecimal representation instead. What follows is a short exposé on the representation of numeric data which you need to know about if you are to understand how computers work.

Binary Data Representation:
Numeric information is stored in computers in the form of groups of binary digits (bits for short), i.e. 0 and 1. For convenience, information is structured in groups of 8 bits (called a byte). Thus, numbers can be represented in direct binary format in which the right-most bit represents 2 to the power of 0; the next one to the left, 2 to the power of 1; the next one, 2 to the power of 2; and so on, until we reach the left-most bit which is 2 to the power of 7 for an 8-bit structure. This can be represented as follows:

2^7+2^6+2^5+2^4+2^3+2^2+2+2^1+2^0

A binary number can be converted to its equivalent decimal by multiplying the value of the appropriate Ith bit (which can be either 0 or 1) with the result of 2^I. For example, the binary number 0001 0101 is equivalent to (0*128)+(0*64)+(0*32)+(1*16)+(0*8)+(1*4)+(0*2)+(1*1)=21 decimal.

It can easily be shown that with n bits available, integer numbers within the range 0 to (2^n–1) can be represented. Therefore,

4 bits (called a nibble) can represent the range 0–15
8 bits (called a byte) can represent the range 0–255
16 bits (two bytes) can represent the range 0–65535.

Note the special case of 10 binary digits which give a maximum of 1024 decimal numbers (0–1023). We represent this by the symbol K so that a 512K computer has (512*1024–1)=524,287 memory locations available. The IBM and compatible computer's microprocessor is capable of addressing a maximum of 640 KBytes of memory starting from location 0 and extending to location 655,359. Of these, 600K are available to the user, as RAM memory, with the rest being used by the system itself.

Hexadecimal Data Representation:
Most operations done by the computer are carried out in binary and although this is easily understood by the computer it causes problems for mere mortals.

For example, the address of memory location 65535 in decimal, is 1111111111111111. Sixteen bits are required to represent all the storage addresses and it is very easy to make mistakes when working with this many digits. The hex numbering system is used to overcome some of these difficulties.

The hex counting system uses base 16 as opposed to base 10 in the decimal and base 2 in the binary system. Counting in 16's is difficult at first, but it does have advantages as you will see later. The value of each column with the different number systems is shown below.

	Value of one unit in column			
	Col 4	*Col 3*	*Col 2*	*Col 1*
Binary	2^3	2^2	2^1	2^0
	(8)	(4)	(2)	(1)
Decimal	10^3	10^2	10^1	10^0
	(1000)	(100)	(10)	(1)
Hexadecimal	16^3	16^2	16^1	16^0
	(4096)	(256)	(16)	(1)

The Hex system requires 16 different digits compared with 10 and 2 in the other systems. It uses 0–9 as in decimal, plus the letters A–F. A list of the first sixteen numbers in each system follows.

Binary	*Decimal*	*Hexadecimal*
0000	00	0
0001	01	1
0010	02	2
0011	03	3
0100	04	4
0101	05	5
0110	06	6
0111	07	7
1000	08	8
1001	09	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

An example, converting a hexadecimal number to a decimal number, will make things clearer. The hexadecimal number 0E00 has a decimal equivalent given by

```
    0  (which is decimal  0) * 4096  =     0
  + E  (which is decimal 14) *  256  =  3584
  + 0  (which is decimal  0) *   16  =     0
  + 0  (which is decimal  0) *    1  =     0
                                        ----
                                        3584
```

The hexadecimal numbering system has an advantage over binary as one hex digit is equivalent to four binary digits (see Table 2 for conversions). Thus, any 8-bit byte of memory can be represented by two hex digits, and any memory address (which requires twenty binary digits) by five hex digits.

Because of the advantage of hex over both binary and decimal systems, it is used for many computing applications. Although it is not necessary to use the hex system, it is essential to understand it if you are to be able to follow how to use the **debug** program or want to understand assembly language programming.

TABLE 2 Hex/Binary/Decimal Conversions

Hex		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Hex	Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	111
0	0000	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
1	0001	1	17	33	49	65	81	97	113	129	145	161	177	193	209	225	241
2	0010	2	18	34	50	66	82	98	114	130	146	162	178	194	210	226	242
3	0011	3	19	35	51	67	83	99	115	131	147	163	179	195	211	227	243
4	0100	4	20	36	52	68	84	100	116	132	148	164	180	196	212	228	244
5	0101	5	21	37	53	69	85	101	117	133	149	165	181	197	213	229	245
6	0110	6	22	38	54	70	86	102	118	134	150	166	182	198	214	230	246
7	0111	7	23	39	55	71	87	103	119	135	151	167	183	199	215	231	247
8	1000	8	24	40	56	72	88	104	120	136	152	168	184	200	216	232	248
9	1001	9	25	41	57	73	89	105	121	137	153	169	185	201	217	233	249
A	1010	10	26	42	58	74	90	106	122	138	154	170	186	202	218	234	250
B	1011	11	27	43	59	75	91	107	123	139	155	171	187	203	219	235	251
C	1100	12	28	44	60	76	92	108	124	140	156	172	188	204	220	236	252
D	1101	13	29	45	61	77	93	109	125	141	157	173	189	205	221	237	253
E	1110	14	30	46	62	78	94	110	126	142	158	174	190	206	222	238	254
F	1111	15	31	47	63	79	95	111	127	143	159	175	191	207	223	239	255

CONTROLLING YOUR SYSTEM

Overview of ANSI.SYS Commands:
Ansi.sys display commands can be used to position the cursor on any part of the screen, change the intensity of the displayed characters, change their colour, or clear part or all of the screen. **Ansi.sys** keyboard commands can be used to redefine keys. For example, you could redefine the function keys so that when you press one a complete command is issued as if it was typed at the keyboard.

ANSI.SYS commands are also called 'escape sequences' because they all begin with the ESCape character (code 27) followed by a left square bracket ([). Commands can also include a numeric or alphabetic code, and each command ends with a different letter. The general form of the command is written as:

```
ESC[<code><letter>
```

where the <code> is a numeric or string value and the ending <letter> identifies the command and is case sensitive (that is, H has a different meaning to h, the former identifying the command that moves the cursor, while the latter sets the display mode). Sometimes, the <code> value might be more than one number or string, in which case it is separated by semicolons. For example,

```
ESC[2J
```

clears the screen, while

```
ESC[2;35H
```

moves the cursor to the 2nd row and 35th column.

ANSI.SYS commands cannot be typed directly into the keyboard because on receiving the ESCape code, MS-DOS cancels the command. Instead, a text editor, such as **edlin** has to be used to create a file with the ESCape codes inserted in command lines. The **ANSI.SYS** commands in the file can then be sent to the console with the use of the **echo** command, or the entire contents of the file can be displayed with the use of the **type** command.

These commands, and the way they are inserted into **edlin**, will be discussed fully later, after we discuss how, using **edlin**, you can build a number of quite ordinary but useful batch files.

Simple Batch Files:
Normally, to complete the implementation your system's hard disc, you need to create a few batch files in a special sub-directory which you might call \BATCH and which will help to run the system efficiently (if you use a \BATCH sub-directory, you must change the PATH command in the **autoexec.bat** file to include the \BATCH sub-directory). For example, you might require to know the exact name of a DOS command. This can be achieved by creating a batch file to display the contents of the DOS sub-directory, whenever the word **dos** is typed. An example of such a batch file (which we will call **dos.bat**), is given below.

```
1:*ECHO OFF
2:*CLS
3:*CD \DOS
4:*DIR/P
5:*CD \
```

In line 3:, the directory is changed to that of \DOS and line 4: causes the contents of the \DOS sub-directory to be displayed using the paging (/P) option. Finally, line 6: returns the system back to the root directory. Thus, typing **dos**, displays the \DOS sub-directory, while typing any external DOS command, invokes the appropriate command.

Some software packages require you to type their name (say, WP for a word processor) in order to activate them. However, most packages also include a second file which is loaded from the first when its name is typed. In such cases you cannot use the PATH command within the **autoexec.bat** file to point to the particular package, as DOS will search for the second file in the root directory. To overcome this, you must use the APPEND command within the **autoexec.bat** file, after the PATH command, as follows:

```
APPEND C:\;C:\WP
```

However if this command is not implemented in your MS/PC-DOS version, you will have to write a special **wp.bat** file to do the same job. The contents of such a file could be:

```
1:*ECHO OFF
2:*CLS
3:*CD \WP
4:*WP
5:*CD \
```

Thus typing **wp**, activates the file which first changes the directory to that of \WP and then loads the file WP. Any other files called from within that file will be found in the correct sub-directory.

Finally, it would be ideal if the language BASIC could be accessed direct from the root directory. However, we can not include a \BASIC sub-directory in the PATH command of the **autoexec.bat** file, as we have done with the \DOS directory, because you might have several versions of the Basic language. For example, two such versions are included in the IBM PC-DOS System disc (BASIC and BASICA; A for advanced), while GWBASIC (which is Olivetti's implementation of the language for use with the compatibles) is included with MS-DOS. Apart from the above versions, you might also have BBCBASIC – a version of BBC-Basic which runs on the IBM and compatible machines.

We can create a rather special batch file, in the \BATCH sub-directory, to access any of these Basic interpreters, provided they are all in the same \BASIC sub-directory, with the following commands in a batch file which we shall call **bas.bat.**

```
1:*ECHO OFF
2:*CLS
3:*CD \BASIC
4:*%1
5:*CD \
```

Note the variable %1 in line 4: which can take the name of any of the Basic languages mentioned above, provided the appropriate name is typed after the batch file name.

For example, typing

```
C:\> BAS GWBASIC
```

at the prompt, starts executing the commands within the batch file **bas.bat**, but substituting GWBASIC for the %1 variable. Thus, line 4: causes entry into GWBASIC, provided it exists in the BASIC directory. Similarly, typing

```
C:\> BAS BASICA
```

causes entry into BASICA. Alternatively, we could use named parameters in batch files which allow definition of replaceable parameters by name instead of by number.

To identify named parameters, we use two percent signs, as follows:

```
%BASTYPE%
```

We can use the SET command to define the named parameter. For example, the command

```
SET BASTYPE=GWBASIC
```

replaces the %BASTYPE% parameter by the filename GWBASIC. The SET command can be used either before the batch file is run, or it can be included within the batch file itself. Thus, the DOS environment variables can be defined as named parameters in a batch file to allow different environments for different applications.

Additional Batch-file Commands:
Apart from the batch-file commands discussed already, there are a number of additional commands which can be useful when writing batch files. These are presented below.

CALL — Calls one batch file from another. The general form of the command is:

CALL filespec

where 'filespec' specifies the drive, sub-directory and name of the batch file to be called. This file must have the extension **.bat**, but must not be included in the 'filespec' part of the CALL command.

The command is used to call a batch file from another batch file.

In the case of pre-v3.3 of DOS, the CALL command can only be used as the last statement of the current file to call another batch file. Return to the first batch file is not possible.

In the case of DOS v3.3 and later, the CALL command can be issued from any place within the current batch file to pass control and execute another batch file. On termination of the called batch file, execution control returns to the calling batch file at the command following the CALL command.

Pipes and redirection symbols must not be used with the CALL command. Batch files that require replaceable parameters can be CALLed. The CALL command can be used to call the current batch file, but care must be taken to eventually terminate execution of the batch file.

FOR	Repeats the specified DOS command for each 'variable' in the specified 'set of items'. The general form of the command is:

```
FOR %%variable IN (set of items) DO command
```

where 'command' can include any DOS command or a reference to the %%var. For example,

```
FOR %%X IN (F.OLD F.NEW) DO TYPE %%X
```

will display F.OLD followed by F.NEW

GOTO label	Transfers control to the line which contains the specified label. For example,

```
GOTO end
---
---
:end
```

sends program control to the :end label

IF	Allows conditional command execution. The general form of the command is:

```
IF [NOT] condition command
```

where 'condition' can be one of

```
EXIST filespec
string1==string2
ERRORLEVEL=n
```

Each of these can be made into a negative condition with the use of the NOT after the IF command.

REM	Displays comments which follow the REM statement

SHIFT — Allows batch files to use more than 10 replaceable parameters in batch file processing. An example of this is as follows:

```
@echo off
:begin
TYPE %1 | MORE
PAUSE
SHIFT
IF EXIST %1 GOTO begin
```

If we call this batch file **display.bat**, then we could look at several different files in succession by simply typing

display file1 file2 file2

as the SHIFT command causes each to be taken in turn.

The System Environment:
The environment is controlled by 'environment variables' which have names and values allocated to them. The SET command can be used to display, change or delete these environment variables. SET typed without parameters displays the current environment. In our case, typing

```
C:\> SET
```

at the prompt will evoke the response

```
COMSPEC=C:\COMMAND.COM
PATH=C:\;C:DOS;C:\BATCH;C:\UTILS
PROMPT=$P$G
```

COMSPEC shows which Command Processor is being used by the system, while PATH and PROMPT display the corresponding commands in your **autoexec.bat** file.

Some software packages require you to SET environment variables to their specifications if the package is to work correctly. However, since there is a limited amount of space allocated to the environment by DOS, space held by these variables in the environment must be freed. This is achieved by typing SET followed by the environment variable and the=sign. For example, to free the environment of the prompt variable we would use SET PROMPT=.

The space put aside by DOS for the environment is 160 bytes initially, but it is then expanded as you define a command path, a system prompt, or create more environment variables, up to a theoretical maximum of just under 1K bytes for DOS v3.0 and v3.1 or 32K bytes for v3.2. However, this environment expansion only takes place if the SET command is used at the command line. If you use an **autoexec.bat** file to set these variables, then the environment is limited to 160 bytes and attempts to increase the number of directories, say, in the PATH command or load memory resident programs, would cause DOS to display the message 'Out of environment space'.

The size of the environment can be increased for MS/PC-DOS v3.0 and later, with the use of the SHELL=configuration command which must be inserted in the **config.sys** file. The general form of the command is:

```
SHELL=COMMAND.COM /E:size /P
```

where

```
    /E:size is the environment size, and
    /P specifies that the shell is to be permanent.
```

The actual 'size' must be written in multiples of 16 bytes, if you are using MS/PC-DOS v3.0 or v3.1 (with a maximum value of 62), or directly in bytes, if you are using v3.2 and later (with a maximum value of 32768).

Environment variables can be used in a batch file to represent the variables' value, provided the environment variable is enclosed in percent signs (i.e. %PATH%).

For example, typing at the command line

```
FOR %N IN (%PATH%) DO ECHO %N
```

will produce the output

```
C:\
C:\DOS
C:\BATCH
C:\UTILS
```

on the screen.

If you intend to include the above line in a batch file, remember that you need to include two percent signs before N (i.e. %%N) in both occurrences in the FOR statement.

As an example of this, let us write a batch file which will display the contents of the **autoexec.bat** and **config.sys** files on the screen. This can, of course, be achieved by using the **type** command at the prompt and specifying the name of each file individually. To achieve the same thing, use **edlin** to create a **show.bat** file in the \BATCH directory, as follows:

```
CLS
FOR %%N IN (\CONFIG.SYS \AUTOEXEC.BAT) DO TYPE %%N
@ECHO OFF
CD\
```

which, from now on, when you type **show.bat**, displays

```
C:\>FOR %N IN (\CONFIG.SYS \AUTOEXEC.BAT) DO TYPE %N

C:\>TYPE \CONFIG.SYS
FILES=20
BUFFERS=30
BREAK ON
COUNTRY=044,437,C:\DOS\COUNTRY.SYS
DEVICE=C:\DOS\ANSI.SYS
DEVICE=C:\DOS\EMM.SYS

C:\>TYPE \AUTOEXEC.BAT
@ECHO OFF
PATH C:\;C:\DOS;C:\BATCH;C:\UTILS
KEYB UK 437,C:\DOS\KEYBOARD.SYS
PROMPT=$P$G
ECHO HELLO ... This is your
VER
```

The ANSI.SYS Console Commands:

The **ansi.sys** commands for controlling the console (display and keyboard) fall into four groups. The first three of these have to do with the control of the display, while the fourth deals with the control of the keyboard. They are:

(a) Cursor control commands,
(b) Erase display commands,
(c) Attribute and mode commands, and
(d) Keyboard control commands.

What follows is a complete summary of all **ansi.sys** console commands appearing under their appropriate category. Each command starts with ESC[(the ESCape character-code 27, followed by a left bracket). The general form of the command is:

`ESC[<code><letter>`

where <code> is a string or numeric value (if more than one, they are separated by semicolons) which identifies the display attribute, display mode, column or row number (or both) to which the cursor is to be moved, the string to be produced when a key is pressed, or the key to be defined. The ending <letter> identifies the command and is case sensitive.

Cursor Control Commands:

Cursor Position	`ESC[#;#H` or `ESC[#;#f` Moves the cursor to the specified position. The first # specifies the row (1-25), while the second # specifies the column (1-80) to which the cursor is to be moved. If either the row or column is omitted, their values, which is 1, is taken. To omit row, but specify column, the semicolon must follow the left bracket. If both row and column are omitted then the cursor moves to the home position which is the upper left corner of the screen.
Cursor Up	`ESC[#A` Moves the cursor up without changing column. The value of # specifies the number of rows by which the cursor is to move up. If the cursor is on the first row, the sequence is ignored. The default value is 1.
Cursor Down	`ESC[#B` Moves the cursor down without changing column. The value of # specifies the number of rows by which the cursor is to move down. If the cursor is on the last row, the sequence is ignored. The default value is 1.

Cursor Right	`ESC[#C` Moves the cursor to the right without changing rows. If the cursor is on the last column, the sequence is ignored. The default value is 1.
Cursor Left	`ESC[#D` Moves the cursor to the left without changing rows. If the cursor is on the first column, the sequence is ignored. The default value is 1.
Save Cursor Position	`ESC[s` Saves the current cursor position. The cursor can be moved to this position later with a Restore Cursor Position command.
Restore Cursor Position	`ESC[u` Restores the cursor position to the value it had when it was last saved with the Save Cursor Position command.
Cursor Position Report	`ESC[#;#R` Reports the current cursor position to the standard input device. The first # specifies the current row, while the second # specifies the current column.
Device Status Report	`ESC[6n` When this command is received, the console driver outputs a Cursor Position Report sequence.

Erase Display Commands:

Erase Display	ESC[2J Erases the screen and moves the cursor to the home position.
Erase Line	ESC[K Erases all text from the current cursor position to the end of the line.

Attribute and Mode Commands:

Set Attribute

`ESC[#;...;#m`

Turns on a display attribute. More than one attribute can be specified provided they are separated by semicolons.

Omitting the value of attribute is equivalent to specifying attribute 0, which turns off all attributes.

Attribute parameter numbers can be any of the following:

Attribute	*Colour*	*Foregrd*	*Backgrd*
0 None	Black	30	40
1 Bold	Red	31	41
4 Underline	Green	32	42
5 Blink	Yellow	33	43
7 Inverse	Blue	34	44
8 Invisible	Magenta	35	45
	Cyan	36	46
	White	37	47

Set Display Mode

`ESC[=#h`

Changes the screen mode and allows line wrap at the 80th column.

A mode parameter number can be one of the following:

Param	*Mode*
0	40x25 b&w
1	40x25 colour on
2	80x25 b&w
3	80x25 colour on
4	320x200 graphics, colour on
5	320x200 graphics, b&w
6	640x200 graphics, b&w
7	Turn on wrap at end of line

Reset Display Mode	`ESC[=#1` The reset mode parameter numbers are the same as those for the Set Display Mode, except that parameter number 7 reset the wrap at the end of a line mode. The I is a lower case letter L.

Keyboard Control Commands:

Define Key	ESC[#;...;#p Assigns one or more characters to be produced when a specified key is pressed. The first # specifies the key to be defined, provided the key is one of the standard ASCII characters with a number from 1 to 127. If the key is a function key, keypad key or a combination of Shift-, Ctrl- or Alt-key and some other key, then two numbers are required separated by a semicolon, the first of which is always 0 and the second taken from the table below. The last # is the character or characters to be produced when a key is pressed. It can be defined as an ASCII code, an extended key code, a string enclosed in double quotes, or any combination of codes and strings separated by semicolons.

Example:

```
ESC[0;68;"dir|sort|more";13p
```

redefines the F10 key so that the directory command is piped to a sort and more commands with a carriage return at the end.

To restore a key to its original meaning, enter a Define Key command sequence that sets the last # equal to the first #.

Example:

```
ESC[0;68;0;68p
```

restores the F10 key to its original meaning.

Extended Key Codes:
The extended key codes used with the **ansi.sys** Define Key command are shown below. Each key can be pressed 'alone', or with the 'shift', 'Ctrl' or 'Alt' keys. A long dash is used in the table to indicate that the key cannot be redefined.

TABLE 3 Extended Codes – Standard ASCII Characters

Key	Alone	Shift	Ctrl-	Alt-
Tab	9	0;15	—	—
-	45	95	—	0;130
0	48	41	—	0;129
1	49	33	—	0;120
2	50	64	—	0;121
3	51	35	—	0;122
4	52	36	—	0;123
5	53	37	—	0;124
6	54	94	—	0;125
7	55	38	—	0;126
8	56	42	—	0;127
9	57	40	—	0;128
=	61	43	—	0;131
a	97	65	1	0;30
b	98	66	2	0;48
c	99	67	3	0;46
d	100	68	4	0;32
e	101	69	5	0;18
f	102	70	6	0;33
g	103	71	7	0;34
h	104	72	8	0;35
i	105	73	9	0;23
j	106	74	10	0;36
k	107	75	11	0;37
l	108	76	12	0;38
m	109	77	13	0;50
n	110	78	14	0;49
o	111	79	15	0;24
p	112	80	16	0;25
q	113	81	17	0;16
r	114	82	18	0;19
s	115	83	19	0;31
t	116	84	20	0;20
u	117	85	21	0;22
v	118	86	22	0;47
w	119	87	23	0;17
x	120	88	24	0;45
y	121	89	25	0;21
z	122	90	26	0;44

Extended Codes – Function and Numeric-keyed Keys

Key	Alone	Shift	Ctrl-	Alt-
F1	0;59	0;84	0;94	0;104
F2	0;60	0;85	0;95	0;105
F3	0;61	0;85	0;96	0;106
F4	0;62	0;87	0;97	0;107
F5	0;63	0;88	0;98	0;108
F6	0;64	0;89	0;99	0;109
F7	0;65	0;90	0;100	0;110
F8	0;66	0;91	0;101	0;111
F9	0;67	0;92	0;102	0;112
F10	0;68	0;93	0;103	0;113
Home	0;71	55	0;119	–
CurlUp	0;72	56	–	–
PgUp	0;73	57	0;132	–
CurLft	0;75	52	0;115	–
CurRgt	0;77	54	0;116	–
End	0;79	49	0;117	–
CurDn	0;80	50	–	–
PgDn	0;81	51	0;118	–
Ins	0;82	48	–	–
Del	0;83	46	–	–
PrtSc	–	–	0;114	–

Using Edlin to Enter ESCape Commands:
The line editor **edlin** can be used to enter ESCape command sequences into a file. The **ESC** character (ASCII 27) is entered by typing **Ctrl-V**, which appears as ^V on the screen, followed by [. Thus, to enter

```
ESC[2J
```

which is the ESCape sequence for 'clear screen', evoke **edlin** and type the appropriate character sequence, as follows:

```
C:\UTILS\> edlin clear
New file
*1i
        1:*^V[[2J
        2:*^C
*e
C:\UTILS\>_
```

Note that we must type two [[, one as part of the ESCape character and the other as required by the ESC[sequence.

To send the ESCape sequence to the display and, in this case, clear the screen, we must use the **type** command as follows:

```
type clear
```

which clears the screen and causes the prompt to reappear on the second row of the display.

Another way of sending the ESCape sequence to the screen, is from within a batch file using the **echo** command. To do this, we must create a **.bat** file and include the command

```
echo^V[[2J
```

in it. The file is then evoked by typing its name only. To eliminate the second prompt which appears on the screen, you must insert, as a first line in the batch file, an **echo off** command.

Note: If you use the **edlin 1** (list) command, you will notice that the ^V[[ESCape sequence has been changed to either [^[(if you are using MS/PC-DOS v3.0 & v3.1), or ^[[(if you are using MS/PC-DOS v3.3 and above).

One advantage of using the **type** command to send the ESCape sequence to the display, is that it is instantaneous. The **echo** method can be very slow, particularly if an elaborate screen is to be built up. For this reason, we shall use the former technique for elaborate screens, and in order to avoid having to type the **type** command, we will write a batch file, say **menu.bat**, which contains the entry **type clear**. Thus, typing **menu** evokes the **menu.bat** file which in turn 'types' the contents of **clear** to the display. Try it. The complete batch file should have the following entries echo

```
echo off
cls
type clear
```

which assumes that both the **menu.bat** and the **clear** files are to be found in the \UTILS sub-directory. Remember, however, that each time you design a new screen file, the fourth line of the above batch file has to be changed to use the name of the newest designed screen. You could, of course, make this general by using **type %1**, which however will require you to provide the screen filename after **menu**. The choice is yours.

We are now in a position to start writing some sample files to provide a simple screen menu design. To do this, you must use **edlin** as explained previously to enter the ESCape code sequences (^V[[for ESC[). As a first attempt, type in the following, where ESC appears in curly ({}) brackets to make identification easier, and call the file **screen1**.

```
1: {ESC}[2J
2: {ESC}[1;30H{ESC}[7mAVAILABLE PACKAGES{ESC}[m
3: {ESC}[3;2H{ESC}[7m1{ESC}[m Basic
4: {ESC}[5;2H{ESC}[7m2{ESC}[m Lotus
5: {ESC}[7;2H{ESC}[7m3{ESC}[m Q&A
6: {ESC}[9;2H{ESC}[7m4{ESC}[m Sage
7: {ESC}[12;2H{ESC}[7mCHOOSE{ESC}[m
8: {ESC}[m
```

Line 1 clears the screen;
Line 2 puts the cursor to row 1, column 30; turns on inverse video (attribute 7) and writes 'AVAILABLE PACKAGES'; turns off inverse video (attribute 0);
Line 3 to line 8 write parts of the menu on the screen.

Type in this program carefully, making sure to check each line before running it. If you make mistakes it is possible that you might have to reboot your computer, as wrong ESCape sequences can cause your computer to hang or do some unexpected things.

Now, run the program (remember that since no **echo** commands were incorporated in the above file, you will have to use either the command **type screen1** or the generalized form of the **menu.bat** file). If all is well, the following should appear on your screen.

```
                    AVAILABLE PACKAGES

1 Basic
2 Lotus
3 Q&A
4 Sage

CHOOSE

C:\UTILS>_
```

The parts of the menu that appear in bold on paper, will in fact appear in reverse video on the screen. Also, note that the cursor reappears under CHOOSE and, in fact, no choice can be made whatsoever on running this program as it stands.

Two things are immediately obvious from the above program: First, there is no method available to 'respond' in some way to the program, by typing in our choice, test the choice and accordingly branch to the required sub-directory to run the chosen program, and second, we don't have any control over the appearance of the cursor – some method of turning the cursor off and on is required.

To achieve the above we must write three small specialized assembly programs (with the **.COM** extension) which call for the use of the **debug** program, which is the subject of the next section.

THE DEBUG PROGRAM

In order to use the **debug** program its command file **debug.com** must be in the currently logged directory or there must be a path to it, as the **debug** program is an external DOS file, in exactly the same way as **edlin.** Again, to simplify matters, copy the **debug.com** file to the \UTILS sub-directory (if you have a hard disc), or to your working floppy, exactly as you did with **edlin.**

Debug can be used to look at memory locations, as well as change such memory locations. It provides a controlled test environment for binary and executable files (files with the **.com** or **.exe** extension). Here, we first start by looking at memory locations of loaded programs, before venturing further afield. In order to demonstrate how this can be done, we will use a four line **test.txt** file which you should create with the use of the **edlin** line editor. The file should contain the following lines of text

```
first line of text
second line of text, edited
third line of text
fourth line of text
```

To start **debug,** type its name followed by the name of the file you want to examine or change. In this case we type

```
C:\UTILS\> debug test.txt
-_
```

provided the file **test.txt** is to be found in the same directory as **debug.** If it does, it causes debug to respond with its own command prompt, in this case a hyphen (-).

The general form of starting **debug** is:

```
debug filespec arguments
```

where 'filespec' can be the full file specification, including drive, directory and filename. The 'arguments' refer to parameters used by the program you want to examine.

When **debug** loads a program into memory, it loads it starting at address 0100 hexadecimal (hex 0100, for short) in the lowest available segment. It also loads the number of bytes placed in memory into the CX register (more about this shortly).

If the filespec is not given when **debug** is started, then it is assumed that you want to do one of the following:

(a) Examine current contents of memory,
(b) Load a program into memory using the **debug Name** or **Load** commands
(c) Load absolute disc sectors into memory with the **Load** command.

The Dump Command:
To examine the contents of memory while using **debug,** type **d** (for dump), followed by 100 (the starting address on which to start the dump). This causes the first 128 bytes of memory starting from hex 100 to be displayed on the screen. In our case, the command

```
-d 0100
```

causes the following block to be displayed on the screen

```
1DC8:0100  66 69 72 73 74 20 6C 69-6E 65 20 6F 66 20 74 65   first line of te
1DC8:0110  78 74 0D 0A 73 65 63 6F-6E 64 20 6C 69 6E 65 20   xt..second line
1DC8:0120  6F 66 20 74 65 78 74 2C-20 65 64 69 74 65 64 0D   of text, edited.
1DC8:0130  0A 74 68 69 72 64 20 6C-69 6E 65 20 6F 66 20 74   .third line of t
1DC8:0140  65 78 74 0D 0A 66 6F 75-72 74 68 20 6C 69 6E 65   ext..fourth line
1DC8:0150  20 6F 66 20 74 65 78 74-0D 0A 1A BB 4E 89 D7 5B    of text....N..[
1DC8:0160  1F C3 C6 06 9E 08 08 B4-00 CD 1A FB 8B FA B0 01   ................
1DC8:0170  E8 4A 00 B4 00 CD 1A FB-3B FA 74 F2 8B FA 8B F7   .J......;.t.....
```

Note that information is divided into three main areas:

Address **Byte value in Hex** **ASCII characters**

```
XXXX:0100  66 69 72 73 74 20 6C 69-6E 65 20 6F 66 20 74 65   first line of te
```

where 'address' refers to the address in memory, starting at hex 1DC8:0100 which is shown above as XXXX:0100 because the first part of the address (the XXXX portion of it) broadly defines the location of it in the computer's memory and is dependent on how much memory is installed and on how many resident programs happen to be loaded at the time. This part of the address will, most likely than not, be different on different computers, therefore it is shown above as XXXX.

Following the address, there is a block of 16 hexadecimal numbers representing the information held in memory so that location 0100, for example, holds the hex value of 66 (which is the ASCII value of the letter f), while location 0108 (just after the hyphen) holds the hex value of 6E (which is the ASCII value of the letter n). The hyphen here serves to divide the block of 16 bytes in half, for easy – location the first half contains bytes 0 to 7, while the second half contains bytes 8 to 15 inclusive.

The last area of the dump is the ASCII characters contained in the file we happen to be examining. Note that any bytes in that portion of memory having a hex value less than 32 are shown by **debug** as periods within this last area. Thus, 0D (carriage return – decimal 13) and 0A (line feed – decimal 10) which occur in memory locations 0112 and 0113, respectively, are shown as .. in the second line of the ASCII character portion of the dump. It is worth your while spending some time examining this dump. For example, try to locate the positions of the 'spaces' in the text which have the hex value of 20.

The dump command can also be used without any parameters (i.e. the starting memory location taken as hex 0100 in our previous example). If this had been done the first time we issued the dump command, after starting **debug**, then dumping would have started at memory location 0100 anyway, as this is the default starting value for a dump of memory. The next time **d** is typed, then the contents of the next 128 bytes of memory are dumped, from hex 0180 to 01FF.

The dump command can also be used to display a specific number of bytes. If this is required, then the command must be followed by the starting and ending address of memory. That is,

```
d start stop
```

Thus, to display the first line of our example, you must type

```
-d 0100 010F
```

Another form of the command, in controlling the number of bytes to be displayed, is by specifying the starting location and the length (L) of the required bytes. For example, the first line of our example can be displayed by typing

```
-d 0100 L 10
```

In the above command, we used uppercase L to specify length, as the lowercase letter could easily by mistaken as the numeral 1. The number of bytes to be displayed above follows L and is hex 10 which is decimal 16.

The Fill Command:
In the dump of the file **test.txt**, we showed the display with certain values after location hex 015A. These values might be different with your computer, because it depends on what happened to be loaded in these locations at the time. We can achieve a more esthetic result with the use of the **f** (for fill) command. The command takes the following form:

```
-f 0100 0180 0
```

which means 'fill memory locations hex 0100 to 0180 with 0'. Do this and verify it by following it with

```
-d 0100
```

Now all the displayed locations hold the hex value 0 and the ASCII character part of the dump contains only periods.

The general form of the fill command is as follows:

```
f range list
```

If a 'range' is specified that contains more bytes than the number of values in the 'list', **debug** uses the 'list' repeatedly until it fills all bytes in the 'range'. If the 'list' contains more values than the number of bytes in the 'range', debug ignores the extra values in the 'list'.

The Load Command:
We can now 'load' our **test.txt** file from the buffer into these zeroed locations with the **L** (for Load) command. Again we use an uppercase letter to avoid confusion by mistaking it for the numeral 1. Thus, typing

```
-L 0100
```

loads our file from the buffer, and to display the result, simply type

```
-d 0100
```

Now you will get a 'cleaner' display of the dump, as the empty memory locations are now filled with 0s.

Note the very last byte of the file in location hex 15A; it contains the value 1A which is what you get when you type **Ctrl-Z**, and represents the end-of-file marker.

The Name Command:
The **n** (for name) command is used to assign a filename to **debug** to use later with the load and write commands. When **debug** is started without specifying a file, the name command must be used in order to set a file. For example,

```
-n file
-L
```

The name command can also be used to supply a program that is to be used by **debug** with information essential to its proper execution. For example, we can use the name command to name a file that requires some data by

```
-n file1.com datafile
-L
```

To take up the earlier example of our file **test.txt** and the requirement of an uncluttered display, we can achieve the same thing by simply typing

```
-f 0100 0180 0
-n test.txt
-L 0100
-d 0100
1DC8:0100  66 69 72 73 74 20 6C 69-6E 65 20 6F 66 20 74 65   first line of te
1DC8:0110  78 74 0D 0A 73 65 63 6F-6E 64 20 6C 69 6E 65 20   xt..second line
1DC8:0120  6F 66 20 74 65 78 74 2C-20 65 64 69 74 65 64 0D   of text, edited.
1DC8:0130  0A 74 68 69 72 64 20 6C-69 6E 65 20 6F 66 20 74   .third line of t
1DC8:0140  65 78 74 0D 0A 66 6F 75-72 74 68 20 6C 69 6E 65   ext..fourth line
1DC8:0150  20 6F 66 20 74 65 78 74-0D 0A 1A 00 00 00 00 00    of text........
1DC8:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

The Enter Command:
The **e** (for name) command allows us to enter data directly into memory as byte values or as a string of characters. The general form of the command is

```
e address list
```

where the values in 'list' replace the contents of one or more bytes starting at 'address'.

Again, assuming that the **test.txt** file has been loaded by **debug**, we can substitute the existing values in memory starting at address hex 0120 with the string "edited by debug", and display the result, with the following commands:

```
-e 0120 "edited by debug"
-d 0100
1DC8:0100  66 69 72 73 74 20 6C 69-6E 65 20 6F 66 20 74 65   first line of te
1DC8:0110  78 74 0D 0A 73 65 63 6F-6E 64 20 6C 69 6E 65 20   xt..second line
1DC8:0120  65 64 69 74 65 64 20 62-79 20 64 65 62 75 67 0D   edited by debug.
1DC8:0130  0A 74 68 69 72 64 20 6C-69 6E 65 20 6F 66 20 74   .third line of t
1DC8:0140  65 78 74 0D 0A 66 6F 75-72 74 68 20 6C 69 6E 65   ext..fourth line
1DC8:0150  20 6F 66 20 74 65 78 74-0D 0A 1A 00 00 00 00 00    of text.......
1DC8:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

The same changes could be achieved by typing the actual values we want to change in hex. For example, typing

```
-e 0120 65 64 69 74 65 64 20 62 79 20 64 65 62 75 67
```

produces the same change as "edited by debug"!

If the 'list' parameter is omitted, then debug displays the address, its contents, and a period, and waits for input.

The Write Command:
The **w** (for write) command writes an area of memory to the file was either last loaded by **debug** or most recently named with the name command. Thus, we can save the changed file of our example above by first naming a file we would like to save the results of the changes and then writing to that file. For example, assuming that the **test.txt** file has been changed with the edit command, we could type

```
-n test1.txt
-w
```

which will save the changes in the **test1.txt** file, leaving the old **test.txt** file unaltered.

The general form of the write command is:

```
w start
```

where 'start' is the starting address in memory from which a number of bytes are written to the file. If 'start' is omitted, **debug** starts at address 0100.

When the write command is executed, **debug** informs you of the total number of bytes (in hexadecimal) it wrote to the file. In this case, the message

```
Writing 005B bytes
```

appears on the screen.

This number is the same as that placed in the CX register when the original file was loaded into memory. In this case, the operation will be correct since we have not changed the actual length of the file. However, had we changed the overall length of the file by, say, appending information to it, then before writing the changes to file, we must change the value held in the CX register to the new length.

Registers:

The Intel Central Processing Units (CPUs) of the processor family that includes the 8086, 8088, 80186, 80286 and 80386 are similar in many respects. All these processors operate internally as 16-bit (two-byte) devices and can, therefore, accept a common set of instructions. In addition, all these processors communicate with the outside world with a 16-bit data bus, with the exemption of the 8088 which operates with an 8-bit data bus which makes it slower.

These CPUs are organized with 14 memory locations, called 'registers', that are 16-bit wide and since they are all located on the processor chip, they can execute instructions very quickly. These registers are subdivided into groups according to the tasks they normally perform. The Table overleaf lists the names, length and normal CPU tasks associated with these registers.

The first four of the CPU registers are referred to as the general purpose registers AX, BX, CX, and DX. These can be used as either 16-bit or 8-bit registers, which is why they are shown in two halves; the high half (H) and the low half (L). Each half can be addressed separately.

Following the general purpose registers are two pointer and two index registers, which serve as pointers to locate data in main memory. These are referred to as SP (stack pointer), BP (Base pointer), SI (source index), and DI (destination index).

Since all the CPU registers are 16-bit long, this means that any such register can only access 2^{16}=65,536 (or 64K) bytes of memory. To overcome this limitation, any of these registers can be combined with an appropriate segment register to address much larger chunks of memory, the actual size being dependent on the total number of combined bits.

For example, SS and SP are combined for stack operations, while CS and IP are combined to locate the next instruction. Mostly, these combinations are arranged within the CPU by default. The maximum addressable memory, when two 16-bit registers are combined end-to-end, corresponds to 232 bytes which is four gigabytes. Such memory addressing is called the 'effective address' with the segment register defining the 64K region of memory and the general register defining the 'offset' from the beginning of the segment.

TABLE 4 Names and Tasks of the CPU Registers

AH	AL	AX, Accumulator
BH	BL	BX, Base
CH	CL	CX, Count
DH	DL	DX, Data
SP		Stack Pointer
BP		Base Pointer
SI		Source Index
DI		Destination Index
DS		Data Segment
ES		Extra Segmant
SS		Stack Segment
CS		Code Segment
IP		Instruction Pointer
Flags		Status flags: NV UP EI PL NZ NA PO NC

The register command allows us to display the names and contents of the registers. To display all the registers, type

```
-r
```

which will cause **debug** to respond with

```
AX=0000  BX=0000  CX=005B  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0100   NV UP EI PL NZ NA PO NC
1DC8:0100 66            DB   66
```

assuming that file **test1.txt** was in memory at the time. Note the contents of the CX register which is 005B, the length of our file.

To change the contents of a register, type the register command, followed by the name of the register. Thus, in the case of the CX register, type

```
-r cx
```

which causes **debug** to repeat the name of the register and the current value held in it (in hex), and then prompt you for a new value by displaying a colon. For example,

```
CX 005B
:_
```

At that point, we can type the new length of the file, in hexadecimal.

Appending to a File:
As an example, let us add the string "Last line addition" to the end of the previous file. We start with address 15A which contains the value 1A representing the Ctrl-Z at the end of the file. This is not needed and can be overwritten. Thus, typing

```
-e 15A "Last line addition"
```

adds 18 (decimal) bytes to the length of the file which was hex 005B (decimal 91) – look up Table 2 in Introduction for conversion of decimal to hex, and vice versa.

Since we have already overwritten the contents of location 15A, the new length is 91−1+18=108 bytes, occupying locations 0100 through to 016B. Now add a carriage return (0D) and a line feed (0A) to the end of the additional line by typing

```
-e 016C 0D 0A
```

which now makes the length to 110 (decimal) bytes or hex 6E.

We now need to change the contents of the CX register, and to this end we type

```
-r cx
```

which causes **debug** to display the present contents of the register and prompt for the change, which we type in as 6E, as follows:

```
CX 005B
:6E
```

Before we write the present contents of memory to file, we can name a new file with the **n** command, say **test2.txt**, by typing

```
-n test2.txt
-w
```

which causes **debug** to respond with

Writing 006E bytes

A dump of the reloaded file is shown below, which verifies what we have discussed above.

```
-d 0100
1DC8:0100  66 69 72 73 74 20 6C 69-6E 65 20 6F 66 20 74 65   first line of te
1DC8:0110  78 74 0D 0A 73 65 63 6F-6E 64 20 6C 69 6E 65 20   xt..second line
1DC8:0120  65 64 69 74 65 64 20 62-79 20 64 65 62 75 67 0D   edited by debug.
1DC8:0130  0A 74 68 69 72 64 20 6C-69 6E 65 20 6F 66 20 74   .third line of t
1DC8:0140  65 78 74 0D 0A 66 6F 75-72 74 68 20 6C 69 6E 65   ext..fourth line
1DC8:0150  20 6F 66 20 74 65 78 74-0D 0A 4C 61 73 74 20 6C    of text..Last l
1DC8:0160  69 6E 65 20 61 64 64 69-74 69 6F 6E 0D 0A 00 00   ine addition....
1DC8:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

The Assemble Command:
The general form of the **a** (for assemble) command is

```
a address
```

where address is the memory location we want to start debug assembling the statement we enter. If the address parameter is omitted, then **debug** starts assembling with the location following the last location assembled. If the assemble command had not been used since starting **debug**, the assembling starts with the location pointed to by CS:IP which is CS:0100 if no file is loaded or if the file loaded is a .COM file. When all statements have been entered, the Return (or Enter) key must be pressed to provide an empty line which signifies the end location for the assembly.

All numeric values must be entered as 1 to 4 hex digits. Prefix assembler mnemonics must be entered in front of the operation codes (called opcodes) to which they refer, but can also be entered on a separate line. In general, a line of source code is divided into the following four sections:

Label	Mnemonic	Operand	Comment

The 'label' is a symbolic reference to the memory location where the next instruction is located, normally used as the target of a jump or subroutine call. A label can contain alphanumeric characters and the underscore character, but the first character must be a letter. A colon is typed at the end of a label to indicate that this label will be referenced only within the current segment of code.

The 'mnemonic' symbolises a CPU instruction, such as MOV (for move), while the 'operand' refers to the operation to be executed, such as AH,02 (AH referring to the destination, while hex 02 is the source). The 'comment' symbolises an explanation of the instruction and must be preceded by a semicolon. Thus, the line

```
begin:  MOV AH,02   ; move hex 02 into register AH
```

represents one possible line of assembler instruction.

Below we list the mnemonics we will be using later in this book, together with their meaning.

TABLE 5 List of Some Common Assembler Mnemonics

ADD	Add destination to source
CMP	Compare destination to source
INT	Call interrupt type
IRET	Interrupt return
JMP	Jump to target
JNZ	Jump if not zero
JZ	Jump if zero
MOV	Move into destination the source

The GO Command:
The **g** (for go) command executes the program in memory. It's general form is:

```
g =address1 address2
```

where 'address1' is the address where **debug** begins execution and changes both the CS and IP registers, while 'address2' set breakpoints which stop the program execution. If both addresses are omitted, then **debug** executes the program normally. If the segment is not specified, then **debug** replaces the value in the IP register with 'address1'. The equal sign must be included with 'address1'. When program execution reaches a breakpoint, the **debug** displays the registers, flags, and decoded instructions of the next instruction ready for execution.

The Go command uses the IRET instruction to cause a jump to the program under test. When a program is completed, then you must reload the program before you can execute it or debug it again.

The Unassemble Command:
The **u** (for unassemble) command, converts memory back to assembly language mnemonics (disassembles bytes) along with address and byte values. The display of a disassembled code looks just like a file ready for assembly. The format is:

```
u address
```
or
```
u range
```

where 'address' is the address at which disassembly starts with the location pointed to by CS:IP. If 'address' is omitted, then **debug** starts converting code after the last location disassembled. If 'range' is omitted, **debug** disassembles 20 hex bytes.

The Quit Command:
The **q** (for quit) command can be used to leave **debug** and return to DOS. The general format of the command is:

```
q
```

This command can be used to exit **debug** without saving any changes made. To save the contents of memory to file, the write command must be issued before the quit command.

There are a lot more commands in **debug**, but what has been presented here is more than enough for our present needs.

WRITING IN ASSEMBLY CODE

We shall now use **debug** to write a few small programs in assembly code which can use DOS directly to produce useful reactions from your computer. You don't need to know how to program in assembler to write these useful utilities, but if you are curious to know what the commands mean, then please refer to Table 1 at the beginning of the book, which lists the ASCII conversion codes, and to Tables 4 and 5 at the end of the previous chapter, which list the respective names of the CPU registers and the assembler mnemonics used in these programs. Interrupts and non-sequential program execution, such as code jumps, are discussed in the next chapter.

It is assumed here that you have configured your system according to the advice given earlier, that is, the command DEVICE=C:\DOS\ANSI.SYS is included in your **config.sys** file, and that PATH C:\;C:\DOS;C:\BATCH;C:\UTILS is incorporated in your **autoexec.bat** file, if you are using a system with a hard disc. The programs created in this book are saved in the \UTILS sub-directory. If, on the other hand, you would prefer to save all created programs on a floppy in the A: drive, then copy to the floppy the **edlin.com** and **debug.com** files from the \DOS sub-directory, of system disc. Thus, you can log into either the \UTILS sub-directory, or the A: drive and start writing in assembly code without further references to other drives or sub-directories.

We start by evoking the **debug** program and typing the commands shown below, as follows:

```
C:\UTILS\> debug
-a 0100
1DC8:0100  mov DL,07
1DC8:0102  mov AH,02
1DC8:0104  int 21
1DC8:0106  int 20
1DC8:0108  <enter>
-r cx
CX 0000
:08
-n bleep.com
-w
Writing 0008 bytes
-g
Program terminated normally
q
C:\UTILS\>_
```

Thus, after invoking **debug**, you type the assemble command

```
a 0100
```

which causes **debug** to respond with

```
XXXX:0100
```

and wait on the same line for your entries. XXXX above is used to indicate that this part of the address will most certainly by different for your computer as it is dependent on the amount of memory available. At this stage you type the move command

```
mov DL,07
```

which moves hex 07 into register DL. Note that hex 07 from Table 1 is in fact the 'bell'. On pressing 'Enter', **debug** responds with

```
XXXX:0102
```

and again waits on the same line for your entries. The move command in the second line

```
mov AH,02
```

selects function 2 (which displays a character) of the DOS interrupt 21 hex, itself being selected by the command

```
int 21
```

in the third line. Finally, the interrupt command

```
int 20
```

in the fourth line, calls a special DOS routine to return control from the current program to DOS. Program flow, as well as interrupts and their various functions, will be discussed shortly. However, you don't need to understand their precise function at this stage, in order not to understand what we have set out to accomplish.

Finally, the assemble command is terminated by pressing the 'Enter' or 'Return' key, shown as

```
<enter>
```

in the fifth line.

Before the program can be written to disc, we need to tell **debug** of its length with the register command

```
r cx
```

which causes **debug** to inform us that the current length is 0000 bytes and prompts us for an entry to which we should respond by typing the actual length which, in this case, is 08.

Now the program is named **bleep.com** with the name command – it is imperative that the extension **.com** is given to assembly language programs. Then, the program is written to disc with the write command which causes **debug** to respond with

```
Writing 0008 bytes
```

The go command can be issued at this point to find out whether the program actually works as expected. If it does, then **debug** responds with the message

```
Program terminated normally
```

at which point we can quit **debug** and can activate the **bleep.com** program by simply typing **bleep.**

However, if you made any mistakes in entering your program into **debug** it is possible that any number of unexpected things might happen when the go command is issued, including no further response from your computer. If this happens, then reset the system and reload into debug the offending program, unassemble it and correct it before saving it and running it again. As an example, we show below what you will see if you use **debug's** unassemble command on the **bleep.com** program.

```
-u 0100

1DE0:0100  B207    MOV  DL,07
1DE0:0102  B402    MOV  AH,02
1DE0:0104  CD21    INT  21
1DE0:0106  CD20    INT  20
1DE0:0108  0000    ADD [BX+SI],AL
```

Note that the entry <enter> in the original program has been changed to what appears in the last line above. Obviously, **debug** is not an easy program to use as an editor to correct long programs, even though it is ideal for assembling short ones.

Creating a Script File:
A better method of creating long assembly language programs is with the use of **edlin** to create what is known as a 'script' file that holds all the information that normally is typed into **debug**, then activate debug with its input redirected to the script file. As an example, we will use **edlin** to create the script file of the same **bleep.com** file.

```
C:\UTILS\> edlin bleep.scr
New file
*1i
     1:* a 0100
     2:*   mov DL,07
     3:*   mov AH,02
     4:*   int 21
     5:*   int 20
     6:*
     7:* r cx
     8:* 08
     9:* n bleep.com
    10:* w
    11:* q
    12:*^C
*e

C:\UTILS\>_
```

Now we can invoke **debug** by typing

```
C:\UTILS\> debug < bleep.scr
```

which will create the desired program automatically. The response of **debug** to this redirection will be:

```
-a 0100
1DC8:0100  mov DL,07
1DC8:0102  mov AH,02
1DC8:0104  int 21
1DC8:0106  int 20
1DC8:0108
-r cx
CX 0000
:08
-n bleep.com
-w
Writing 0008 bytes
-q

C:\UTILS\>_
```

In this way, any errors in the program can be put right by first using **edlin** to correct the **.scr** file, and then starting again **debug** with its input redirected to the corrected script file.

Control of Program Flow:
The microprocessor is responsible for the correct control of program flow. This is achieved by repeatedly

(a) fetching instructions from memory, and
(b) executing them.

In the absence of any program jumps or calls to subroutines, these instructions are executed sequentially.

A register called the Program Counter (PC) controls which instructions are fetched and at any given time this register contains the memory location holding the instruction being executed. After execution of the first instruction, the microprocessor increments the Program Counter by one and fetches the next instruction to be executed. This process continues until all the instructions in the program are executed.

Non-sequential Program Execution:
The 'jump' instruction performs an unconditional jump in program execution.

An example of this is given below. Start **debug** in the usual way, then type the following instructions:

```
-f 0100 0200 0
-a 0100
1DC8:0100 mov AL,05
1DC8:0102 jmp 0106
1DC8:0104 mov AL,071
1DC8:0106 mov [0110], AL
1DC8:0109
1DC8:0109
-
```

The program first puts the hex number 05 into the low part of the AX register (AL), and then program execution jumps unconditionally to location 0106, thus avoiding the second **mov** instruction in location 0104. Then, in order to prove that this jump command has been executed correctly, the instruction in location 106 forces the contents of the AL register to be copied into memory address [0110]. Program execution can be started by typing the **go** command, namely, g =0100 0109.

Using Debug's Trace Command:
Debug's t (for trace) command is used to execute a program, instruction by instruction. Thus, typing

```
-t =0100
```

causes **debug** (if you include the=sign) to display register information after each trace command, as follows:

```
-t =0100

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0102   NV UP EI PL NZ NA PO NC
1DC8:0102 EB02          JMP 0106
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0106   NV UP EI PL NZ NA PO NC
1DC8:0106 A21001        MOV [0110],AL                         DS:0110=00
-t

AX=0005  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0109   NV UP EI PL NZ NA PO NC
1DC8:0109 0000          ADD [BX+SI],AL                        DS:0000=CD
-
```

In this case, the value 05 is first moved into register AL. The next instruction is at location 0102 which however forces a jump to location 0106. The contents of the lower part of the AX register are still holding the value 05.

Finally, to prove that the value 05 has indeed been moved into address 0110, type

```
-d 0100
1DC8:0100  B0 05 EB 02 B0 07 A2 10-01 00 00 00 00 00 00 00   ................
1DC8:0110  05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-q
```

which shows that value 05 has been moved into location 0110.

Conditional Program Jumps:
Conditional jumps in program execution can be achieved by first setting the condition and then using the **cmp** (for compare) command. This command compares two parameters and if the difference between them is zero, then a special flag is set. This is the fifth flag shown at the bottom of Table 4, which is marked NZ (for not zero). Flags are registers which have only two states; they either 'set' or 'not set'. The **cmp** command sets the zero flag which then reads ZR (for zero). If the zero flag has not been set, it appears as NZ.

The following example will help the illustrate the above points. Evoke **debug** and type the following instructions:

```
-f 0100 0200 0
-a 0100
1DC8:0100 mov AL,05
1DC8:0102 cmp AL,05
1DC8:0104 jnz 0108
1DC8:0106 mov [0110],AL
1DC8:010B
```

Then use the trace command to see the program execute one instruction at a time. This will also display the state of the zero flag. The display should appear as follows:

```
-t =0100

AX=0005  BX=0000  CX=000B  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0102   NV UP EI PL NZ NA PO NC
1DC8:0102 3C05          CMP AL,05
-t

AX=0005  BX=0000  CX=000B  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0104   NV UP EI PL ZR NA PE NC
1DC8:0104 7502          JNZ 0108
-t

AX=0005  BX=0000  CX=000B  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0106   NV UP EI PL ZR NA PE NC
1DC8:0106 B007          MOV AL,07
-t

AX=0007  BX=0000  CX=000B  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=1DC8  ES=1DC8  SS=1DC8  CS=1DC8  IP=0108   NV UP EI PL ZR NA PE NC
1DC8:0108 A21001        MOV [0110],AL                           DS:0110=00
-
```

Note that the zero flag appears as NZ (being not set), immediately after the first trace command which executes the first **mov** command, moving the value 05 into AL. After the second trace command, the zero flag is set and it appears as ZR once the **cmp** command is executed. The IP (instruction pointer) register holds the location of the next executable statement, which in this case confirms that since the result of the comparison is zero and the conditional program jump only takes place if the difference is 'not zero', then the next instruction to be executed is to be found in location 0106 which moves the value 07 into AL. To confirm this, dump the contents of the relevant locations, as follows:

```
-d 0100
1DC8:0100  B0 05 3C 05 75 02 B0 07-A2 10 01 00 00 00 00 00   ..<.u...........
1DC8:0110  07 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
1DC8:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-q
```

which shows that value 07 has been moved into location 0110.

Now use **debug** to change the instruction starting in location 0104, as follows:

```
-a 0104
1DC0:0104 jz 0108
1DC0:0106
```

and then use the trace command again to confirm that the program will now skip the instruction in location 0106, thus placing the hex value 05 into location 0110.

Interrupts:
Interrupts provide a way for I/O devices to communicate with the CPU. An interrupt informs the processor that an external device needs attention, which causes the processor to suspend its current activity and respond to it. On receipt of an interrupt, the processor finishes executing the last instruction, saves the address of the next instruction on the stuck (a special contiguous memory block, the location of which is to be found in the SP (stuck

pointer) register), then jumps to the special interrupt handling subroutines which are to be found in certain parts of the computer's memory, executes the appropriate one and returns to the suspended program by fetching the address of the next instruction from the stuck.

The Intel 8088 and 8086 family of processors can address one megabyte of memory by combining a general register with a segment register. Different parts of this memory have been apportioned to different activities with some activities being in ROM (Read Only Memory) and containing permanent instructions for the operation of the computer, while other activities, such as user's programs and their data are held in RAM (Random Access Memory).

The memory map of an IBM PC is shown below, with each of the 16 possible different segments containing 64 Kbytes. The address of each 64 Kbyte segment begins with a different hexadecimal digit (0–9) or letter (A–F), with the first two referring to the 64 Kbyte segment, and the second two to the offset within each segment.

TABLE 6 Memory Map of the IBM PC

Address	Description
0000	BIOS interrupts
0080	DOS interrupts
0040	BIOS data area
0050	DOS & Basic data area
A800	Enhanced graphics
B000	Monochrome adapter
B800	Graphics adapter
C800	Hard Disc ROM
F600	ROM Basic
FE00	ROM BIOS

Memory locations from 0000 to 9FFF, which constitutes 640KB, is allocated as RAM working space. Segments A800, B000 and B800 contain RAM allocated respectively to enhanced-graphics video memory, and the video screens. Segments C800, F600 and FE00, contain ROM and is allocated respectively to the operation of the hard disc, Basic and instructions for the power up self-test and operation of peripherals.

The first 1024 bytes of RAM memory, known as the *interrupt vector table,* contain the 256 interrupt vectors which provide entry points into subroutines residing elsewhere in memory. These subroutines communicate directly with peripherals through registers and their addresses are numbered from 0 to FF. Since these vectors are located in memory, any program can use them to demand service from the appropriate subroutine.

The interrupt vectors are organized under a priority scheme and can be grouped into three basic categories, as follows:

(a) Internal hardware interrupts occupy the lowest part of the 1024 bytes of system memory from hexadecimal 00 to 1F, with interrupt levels running from hex 00 to 0D. These are generated by certain events during program execution, such as encountering an invalid opcode. The assignment of such events to the appropriate interrupt number is wired into the processor and is unmodifiable.

(b) External hardware interrupts occupy certain areas within the first 1024 bytes of system memory other than hex ranges 00 to 1F and 20 to 3F. The external hardware interrupts are triggered by co-processors or controllers of peripheral devices. They are not wired directly to the CPU, but are channeled through a special Programmable Interrupt Controller device, PIC for short, which is controlled by the CPU through a set of I/O ports. Different peripheral devices are assigned to their corresponding interrupt levels, the assignment being made by the manufacturers of the equipment and thus being unmodifiable by software.

(c) Software interrupts can be triggered by any program by simply issuing the instruction

```
INT N
```

where N is the interrupt level number which then generates a call to the address of the appropriate subroutine. Interrupts with hexadecimal numbers from 20 to 3F, are used by DOS to communicate with its modules and with application programs. Other interrupts are used by the ROM BIOS, by application software for various purposes, or system drivers. The table overleaf lists the interrupt vector type and its function.

TABLE 7 Software Interrupt Vectors for the IBM PC

Vector	Function
05	Print screen
08	Timer
09	Keyboard scan code
0B	Asynchronous comms port controller 1
0C	Asynchronous comms port controller 0
0D	Hard disc controller
0E	Floppy disc controller
0F	Printer controller
10	Video screen driver
11	Equipment configuration check list
12	Memory size check
13	Hard/Floppy disc driver
14	Serial comms port driver
15	Cassette I/O, AT auxiliary functions
16	Keyboard driver
17	Printer driver
18	ROM Basic
19	Bootstrap loader
1A	Read/Set time clock
1B	Ctrl-Break handler
1C	Timer control
1D	Video parameter table
1E	Disc parameter table
1F	Graphics character table
20	Terminate COM program
21	General DOS services
22	Program terminate code
23	Ctrl-C code
24	Error code
25	Absolute disc read
26	Absolute disc write
27	Terminate but stay resident
28-2E	Reserved for DOS
2F	Print spooler
30–3F	Reserved for DOS
40	Floppy disc driver
41	Hard disc parameter table

The most important DOS interrupt is hex 21 because it performs a number of useful operations. These operations as categorized and are given a 'function' number. To use these functions, the function number must be placed in register AH. Once this is done, and other registers are established as required, the command INT 21 is given. On return, data may be available in one or more registers.

TABLE 8 The Functions of DOS Services Interrupt 21

AH value	Function description
00	Terminate program and return to DOS
01	Read and display keyboard input character
02	Write character to video screen
03	Read from serial port
04	Write to serial port
05	Write to printer port
06	Direct keyboard input and video output
07	Read keyboard input without echo or Break detection
08	Read keyboard input without echo but with Break detection
09	Display a string of characters
0A	Read keyboard buffer
0B	Keyboard input status
0C	Reset input buffer and invoke keyboard input
0D–24	Disc operations, but with 18 & 1D-20 reserved
25	Set machine interrupt vector to point to an interrupt handling routine
26	Create program segment prefix
27–29	File operations
2A–2D	Fetch and set system date and time
2E	Set verify flag
2F	Fetch disc transfer area address
30	Fetch DOS version number
31	Terminate but stay resident
32	Reserved
33	Fetch or set Ctrl-Break flag
34	Reserved
35	Fetch interrupt vector
36	Fetch free disc space
37	Reserved
38	Fetch or set country
39–43	File and directory operations
44–47	File and device-driver control information
48	Allocate block of memory
49	Release block of memory
4A	Change size of allocated memory
4B	Load and execute a program
4C	Terminate a program and return to DOS
4D	Fetch return code
4E–5D	File operations, but with 50–53, 55 and 5D reserved
5E	Fetch and set printer setup
5F–63	Redirection and address operations, but with 60–61 reserved

THE FINAL ASSEMBLAGE

Creating Interactive Batch Files:
In order to make batch files interactive, we need to create a small program which 'responds' to the keyboard keys most recently pressed. This is a bit similar to the INKEY command in high level computer languages that reads a character from the keyboard.

Normally, when a key is pressed, a code representing that key is sent to DOS for translation and subsequent display. However, DOS also stores the value of this code in a part of memory which can be accessed and is normally referred to as the 'errorlevel'. The key codes of both the standard ASCII and extended ASCII characters were discussed earlier and are listed in Tables 1 and 2, respectively.

Because the first number of the two-number value representing the extended key codes is always 0, DOS sets errorlevel to the second number. This, inevitably produces some duplication between standard and extended key codes (for example, the numeric key 0, Alt-b and Shift-Ins all set errorlevel to 48), but we can put up with it because the keys responsible are unrelated.

To create **respond.com**, use **edlin** to create its script file as follows:

```
C:\UTILS\> edlin respond.scr

     1:* a 0100
     2:*   mov AH,07
     3:*   int 21
     4:*   cmp AL,00
     5:*   jnz 010C
     6:*   mov AH,07
     7:*   int 21
     8:*   mov AH,4C
     9:*   int 21
    10:*
    11:* r cx
    12:* 10
    13:* n respond.com
    14:* w
    15:* q

C:\UTILS\>_
```

Now, we can invoke **debug** by typing

```
C:\UTILS\> debug < respond.scr
```

which will create the desired program automatically.

We can then rewrite the **menu.bat** file (calling it **menu1.bat**) to incorporate the **respond.com** file as follows:

```
echo off
cls
  type screen1
:GETKEY
  respond
  if errorlevel 53 goto GETKEY
  if not errorlevel 49 goto GETKEY
```

where **screen1** was created earlier in this book, using **edlin**, and has the following contents:

```
    1: {ESC}[2J
    2: {ESC}[1;30H{ESC}[7mAVAILABLE PACKAGES{ESC}[m
    3: {ESC}[3;2H{ESC}[7m1{ESC}[m Basic
    4: {ESC}[5;2H{ESC}[7m2{ESC}[m Lotus
    5: {ESC}[7;2H{ESC}[7m3{ESC}[m Q&A
    6: {ESC}[9;2H{ESC}[7m4{ESC}[m Sage
    7: {ESC}[12;2H{ESC}[7mCHOOSE{ESC}[m
    8: {ESC}[m
```

On running **menu1.bat** now, we get the following display:

AVAILABLE PACKAGES

1 Basic

2 Lotus

3 Q&A

4 Sage

CHOOSE

—

with the cursor appearing under **CHOOSE** rather than next to it. Nevertheless, the program now responds only to inputs 1 to 4 inclusive, which make the prompt reappear.

Note that 'errorlevel' is checked backwards; the first **if** command checks whether 'errorlevel' is greater than or equal to the specified number which, in this case, excludes all codes greater than or equal to 53. The second **if** command in the batch file first checks to see whether 'errorlevel' is greater than or equal to 49, but then the 'not' in the statement inverts the logic which now has the effect of checking to see whether 'errorlevel' is less than the specified number, which in this case is 49. In this way the batch file returns command to DOS only if keys 1 to 4 are typed which corresponds to key codes 49 to 52 inclusive.

We can improve the batch file **menu1.bat** to actually respond by telling us which key was pressed. To do this, modify the file (calling it **menu2.bat**) to include the following:

```
echo off
cls
:AGAIN
  type screen1
:GETKEY
  respond
  if errorlevel 53 goto GETKEY
  if errorlevel 52 goto FOURTH
  if errorlevel 51 goto THIRD
  if errorlevel 50 goto SECOND
  if errorlevel 49 goto FIRST
  if errorlevel 48 goto QUIT
  if not errorlevel 48 goto GETKEY
:FOURTH
    echo You have typed 4
    respond
    goto AGAIN
:THIRD
    echo You have typed 3
    respond
    goto AGAIN
:SECOND
    echo You have typed 2
    respond
    goto AGAIN
:FIRST
    echo You have typed 1
    respond
    goto AGAIN
:QUIT
    cls
    bleep
    echo Bye
```

Now, the batch file responds appropriately to the keys you type, with the 0 (zero) acting as the 'quit' key. Note the use of the **respond** program as a 'pause' command (without the latter's message to the user), which passes control to the **goto** command only after a key (any key) is pressed.

Controlling the Cursor:
We can improve the appearance of the previously written menu batch files by incorporating two assembly language programs which control the cursor. The first program is designed to turn the cursor off, so that it does not appear in unwanted areas on the screen, while the second is designed to turn the cursor back on again.

Now use **edlin** to first write the script file **cursoff.scr**, to turn the cursor off, with the following contents:

```
 1:* a 0100
 2:*   mov AH.01
 3:*   mov CH,20
 4:*   int 10
 5:*   int 20
 6:*
 7:* r cx
 8:* 08
 9:* n cursoff.com
10:* w
11:* q
```

then write the script file curson.scr, to turn the cursor on, with the following contents:

```
 1:* a 0100
 2:*   mov AH,0F
 3:*   int 10
 4:*   cmp AL,07
 5:*   jz  010D
 6:*   mov CX,0607
 7:*   jmp 0110
 8:*   mov CX,0B0C
 9:*   mov AH,01
11:*   int 10
12:*   int 20
13:*
14:* r cx
15:* 16
16:* n curson.com
17:* w
18:* q
```

Now, use **debug** with its input redirected to the script file **cursoff.scr**, to create the **cursoff.com** program, as follows:

```
C:\UTILS\>debug < cursoff.scr
```

followed by the reactivation of debug with its input redirected to the script file curson.scr to create the **curson.com** program.

To demonstrate how these two programs can be used to enhance batch files, edit the contents of the batch file **menu1.bat** (calling it **menu3.bat**) incorporating the following changes:

```
echo off
cls
  type screen1
  cursoff
:GETKEY
  respond
  if errorlevel 53 goto GETKEY
  if not errorlevel 49 goto GETKEY
  curson
```

On running this batch file, you will see that now the cursor is removed from the screen and does not reappear until you type the correct information. Note that both these programs (as indeed all the programs we have created using **debug**) can be used by themselves. Thus typing **cursoff** will make the cursor disappear from the screen, while typing curson makes it reappear. As an exercise, try to use **cursoff** and **curson** into the **menu2.bat** file. Call the result **menu4.bat.**

Designing a Menu Screen:

We can now use all the expertise gathered so far to design a menu screen which is pleasing to the eye. As all the menu titles appear within a boxed area, the box-drawing characters needed for this are given below for convenience, even though they also appear in Table 1.

Now use **edlin** to create **screen2**, remembering that the {ESC} part of each line is entered in **edlin** using the ESCape code sequence ^V[, followed by the square bracket ([) shown in the text after {ESC}. Note that line 2 is shown in the text foreshortened; it should, in fact, be 29 = characters long before the {ESC} sequence preceding the message, and 30 = characters long after the message. Similarly, at the bottom of the file, the corresponding number of the = character is 30 before and 31 after the message. The number of the character making up the single lines should be 76 each.

```
╔Alt-201          ═Alt-205          ╗Alt-187

╟Alt-199          ─Alt-196          ╢Alt-182

╚Alt-200          ║Alt-186          ╝Alt-188
```

```
{ESC}[2J
╔═══════ {ESC}[7mAVAILABLE PACKAGES{ESC}[m ═══════╗
{ESC}[3;1H ╟──────────────────────────────────────╢
║{ESC}[4;12H{ESC}[7mA{ESC}[m Program 01
║{ESC}[5;12H{ESC}[7mB{ESC}[m Program 02
║{ESC}[6;12H{ESC}[7mC{ESC}[m Program 03
║{ESC}[7;12H{ESC}[7mD{ESC}[m Program 04
║{ESC}[8;12H{ESC}[7mE{ESC}[m Program 05
║{ESC}[9;12H{ESC}[7mF{ESC}[m Program 06
║{ESC}[10;12H{ESC}[7mG{ESC}[m Program 07
║{ESC}[11;12H{ESC}[7mH{ESC}[m Program 08
║{ESC}[12;12H{ESC}[7mI{ESC}[m Program 09
║{ESC}[4;34H{ESC}[7mJ{ESC}[m Program 10
║{ESC}[5;34H{ESC}[7mK{ESC}[m Program 11
║{ESC}[6;34H{ESC}[7mL{ESC}[m Program 12
║{ESC}[7;34H{ESC}[7mM{ESC}[m Program 13
║{ESC}[8;34H{ESC}[7mN{ESC}[m Program 14
║{ESC}[9;34H{ESC}[7mO{ESC}[m Program 15
║{ESC}[10;34H{ESC}[7mP{ESC}[m Program 16
║{ESC}[11;34H{ESC}[7mQ{ESC}[m Program 17
║{ESC}[12;34H{ESC}[7mR{ESC}[m Program 18
║{ESC}[4;56H{ESC}[7mS{ESC}[m Program 19
║{ESC}[5;56H{ESC}[7mT{ESC}[m Program 20
║{ESC}[6;56H{ESC}[7mU{ESC}[m Program 21
║{ESC}[7;56H{ESC}[7mV{ESC}[m Program 22
║{ESC}[8;56H{ESC}[7mW{ESC}[m Program 23
║{ESC}[9;56H{ESC}[7mX{ESC}[m Program 24
║{ESC}[10;56H{ESC}[7mY{ESC}[m Program 25
║{ESC}[11;56H{ESC}[7mZ{ESC}[m Program 26
║{ESC}[12;56H{ESC}[7m@{ESC}[m Ret to DOS
{ESC}[4;79H ║
{ESC}[5;79H ║
{ESC}[6;79H ║
{ESC}[7;79H ║
{ESC}[8;79H ║
{ESC}[9;79H ║
{ESC}[10;79H ║
{ESC}[11;79H ║
{ESC}[12;79H ║
{ESC}[13;1H ╟──────────────────────────────────────╢
╚═══════ {ESC}[7mTYPE A CHARACTER{ESC}[m ═══════╝
{ESC}[m
```

When you have finished entering the information into **edlin**, you can test your creation by typing

```
type screen2
```

which should cause the following menu to appear on the screen.

```
════════════════════ AVAILABLE PACKAGES ════════════════════

   A Program 01       J Program 10       S Program 19
   B Program 02       K Program 11       T Program 20
   C Program 03       L Program 12       U Program 21
   D Program 04       M Program 13       V Program 22
   E Program 05       N Program 14       W Program 23
   F Program 06       O Program 15       X Program 24
   G Program 07       P Program 16       Y Program 25
   H Program 08       Q Program 17       Z Program 26
   I Program 09       R Program 18       @ Ret to DOS

═════════════════════ TYPE A CHARACTER ═════════════════════
```

This menu screen needs to be controlled by an appropriate batch file which we will now create using **edlin.** We will call this batch file **menu5.bat** and we will include in it the following commands:

```
echo off
cls
:AGAIN
  type screen2
  echo {ESC}[7mCHOOSE{ESC}[m {ESC}[5m_{ESC}[m
  cursoff
:GETKEY
  respond
  if errorlevel 91 goto GETKEY
  if not errorlevel 64 goto GETKEY
  if errorlevel 64 if not errorlevel 65 goto QUIT
  echo You have requested a package
  pause
  goto AGAIN
:QUIT
  curson
  cls
```

Note the line with the {ESC} code sequence within the above batch file. It displays the word CHOOSE in inverse video and then imitates the presence of a cursor by flashing the underscore character right next to it.

Also, note that the **if** commands within the batch file are only tested against capital letters codes only. In other words, the input to the file is case sensitive. You will only get the message

```
You have requested a package
Press any key when ready ...
```

if you choose uppercase letters. The program does not respond to lower case letters or any other keyboard characters. To return to DOS type the key marked '@'.

Being Security Conscious:
You can use the information given so far, to obtain some modest security for your system. For example, if you work in an environment where you often need to walk away from your computer for lengthy periods of time and want to keep prying eyes away from your work, then could devise a simple batch file to give you that security without having to switch off your computer. In any case, your computer will last much longer if you avoid switching it on and off too many times during a working day.

Again, use **edlin** to write a batch file (call it **sleep.bat**), with the following contents:

```
@echo off
cls
  cursoff
:GETKEY
  respond
  if errorlevel 114 goto GETKEY
  if not errorlevel 113 goto GETKEY
curson
```

Once the **sleep.bat** file is activated, you can only reawaken your computer by typing **q** or **Alt-F10**. You could, of course, choose your own key combination from Table 3.

However, the **sleep.bat** file could be interrupted by pressing repeatedly fast the **Ctrl-C** or **Ctrl-Break** keys. Pressing a single **Ctrl-C** or **Ctrl-Break**, will have no effect because of the way the **respond.com** file was written, in the first place, with function 07 of interrupt 21, which does not check for **Ctrl-Break**. But, if while one **Ctrl-Break** is being processed, another one is issued fairly rapidly behind the first, then the batch file would most likely be interrupted.

Suspending Ctrl-Break:
To overcome the above limitation, two assembly language programs will be created which we will call **brkoff.com** and **brkon.com**, for break-off and break-on, respectively. The DOS service interrupt 21 with function 33 will be used to determine the current status of the operating system's **Ctrl-C** or **Ctrl-Break** checking flag. When fetching the status of this flag we let AL=00, while when setting the status of the flag, we let AL=01. Now use **edlin** to first create the script file for the break-off situation, with the following contents:

```
 1:* a 0100
 2:* mov AH,33
 3:* mov AL,01
 4:* mov DL,00
 5:* int 21
 6:* int 20
 7:*
 8:* r cx
 9:* 0A
10:* n brkoff.com
11:* w
12:* q
```

then to create the script file for the break-on situation, with the following contents:

```
 1:* a 0100
 2:* mov AH,33
 3:* mov AL,01
 4:* mov DL,01
 5:* int 21
 6:* int 20
 7:*
 8:* r cx
 9:* 0A
10:* n brkon.com
11:* w
12:* q
```

Finally, use **debug** successively with redirection, first to the break-off script file, then to the break-on script file, to produce the two programs **brkoff.com** and **brkon.com.**

Now create the controlling batch file (call it **secure.bat**), with the following contents:

```
@echo off
cls
  cursoff
  brkoff
:GETKEY
  respond
  if errorlevel 114 goto GETKEY
  if not errorlevel 113 goto GETKEY
  curson brkon
```

When **secure.bat** is activated, **Ctrl-Break** has no effect and cannot be used to terminate the batch file.

By also inserting the command secure in an appropriate place within the **autoexec.bat** file, it is possible to make your system even more secure. Its position would depend on whether or not the actual batch file **secure.bat** and all the **.com** programs it uses are to be found in the root directory or not. If not, then the command **secure** must be inserted after the PATH declaration in which case it is possible to use **Ctrl-Break** early enough to terminate the **autoexec.bat** file before the **secure** command is reached.

APPENDIX A
SYSTEM CONFIGURATION

If you are using a hard disc, then it is assumed that you have structured it in such a way as to hold all the DOS external command files in the sub-directory \DOS, all the batch files in the sub-directory \BATCH, and that you would like to hold all the utility programs we develop in this book, in a sub-directory called \UTILS. This supposition is reflected in the full filespec given for **ansi.sys** within the configuration file and the PATH command within the **autoexec.bat** file.

The CONFIG.SYS File:
This file allows you to configure your computer to your needs, as commands held in it are executed during boot-up. The easiest way to create or amend this system file is with the use of the line editor edlin.

Now use **edlin** to edit your **config.sys** file in such a way as to include the following commands:

```
1:*FILES=20
2:*BUFFERS=30
3:*BREAK=ON
4:*COUNTRY=044
5:*DEVICE=C:\DOS\ANSI.SYS
```

Users of MS/PC-DOS v3.3 and above, see below for different COUNTRY=entry.

Following is a list of the commands that you can include within the **config.sys** file which MS-DOS supports. However, do remember that any changes made to this file only take effect after rebooting which can be achieved by pressing the three keys marked **Ctrl, Alt** and **Del** simultaneously.

BREAK	By including the command BREAK=ON in the **config.sys** file, you can use the key combination **Ctrl-C** or **Ctrl-Break**, to interrupt DOS I/O functions.
BUFFERS	DOS allocates memory space in RAM, called buffers, to store whole sectors of data being read from disc. The default number of buffers is 2, each of 512 bytes of RAM. If more data are required, DOS first searches the buffers before searching the disc, which speeds up operations.

COUNTRY

DOS displays dates according to the US format which is month/day/year. To change this to day/month/year, use the command

```
COUNTRY=044
```

where 044 is for U.K. users. The default value is 001, for the USA.

Users of a hard disc with PC-DOS 3.3 should enter this statement as

```
COUNTRY=044,437,C:\DOS\COUNTRY.SYS
```

where 437 is the code page of pre-3.3 versions of DOS and **country.sys** is to be found in the \DOS sub-directory.

CODEPAGE

This command is to be found in PC/MS-DOS versions 3.3 and later. The table that DOS uses to define a character set is called a code page. Thus include the command

```
CODEPAGE=437
```

where 437 is the code page definition of pre-3.3 versions of DOS.

DEVICE

DOS includes its own standard device drivers which allow communication with your keyboard, screen and discs. However, these drivers can be extended to allow other devices to be connected by specifying them in the **config.sys file**. Example of these are:

```
DEVICE=ANSI.SYS
```

which loads alternative screen and keyboard drivers for ANSI support – features of which are required by some commercial software.

```
DEVICE=KBOARD.SYS /xx
```

allows IBM compatibles under pre-v3.3 MS-DOS the use of different keyboards. For UK users the /xx option is /UK.

MS-DOS v3.3 requires the entry

```
KEYB UK 437,C:\DOS\KEYBOARD.SYS
```

in the **autoexec.bat** file, instead of the DEVICE=entry in the **config.sys** file.

In the case of IBM's PC-DOS, the keyboard layout is contained in the **keybuk.com** file which is activated by the entry

```
KEYBUK
```

in the **autoexec.bat** file.

```
DEVICE=MOUSEAnn.SYS
```

allows IBM compatibles the use of specific mouse devices.

If an IBM mouse is fitted to your computer, then a file called **mouse.com** is required. This file is made available by IBM on purchase of the mouse and should be copied to the \DOS sub-directory. The file can then be activated by the entry

```
MOUSE
```

in the **autoexec.bat** file, as opposed to using the DEVICE=entry in the **config.sys** file.

FILES DOS normally allows 8 files to be opened at a time. However, some software such as relational databases, might require to refer to more files at any given time. The maximum allowable is 99, but 20 is usually quite adequate.

The AUTOEXEC.BAT File:
This is a special batch file that DOS looks for during the last stages of the booting up process and if it exists, the commands held in it will be executed. One such command is the KEYBxx which configures keyboards for the appropriate national standard, with xx indicating the country. For the U.K., the command becomes KEYBUK, and you will need to execute it if your keyboard is marked with the double quotes sign on the 2 key and/or the @ sign over the single quotes key and/or the £ sign over the 3 key.

Possible contents of the **autoexec.bat** file are as follows:

```
1:*ECHO OFF
2:*CLS
3:*PATH C:\;C:\DOS;C:\BATCH;C:\UTILS
4:*KEYBUK
5:*PROMPT $P$G
6:*ECHO HELLO ... This is your
7:*VER
```

Users of a hard disc with MS/PC-DOS 3.3 should enter the KEYBUK statement as

```
4:*KEYBUK 437,C:\DOS\KEYBOARD.SYS
```

where 437 is the code page of pre-3.3 versions of DOS and **keyboard.sys** is to be found in the \DOS sub-directory. As mentioned previously under the COUNTRY section, in PC-DOS 3.3 the extended IBM character set has been changed slightly to accommodate several versions of it by offering several choices on the characters displayed or printed. Each such version is referred to by a specific code page number which defines the character set to be used. If you intend to use any other code page than 437, then you should refer to your DOS reference guide.

The commands in the above **autoexec.bat** file have the following effect. First echoing is switched off, but only after executing the ECHO OFF command and thus, to clear the screen of the displayed command, we employ the CLS command (for v3.3 users these first two commands could be replaced by @ECHO OFF. The use of the @ in front of any command eliminates the echoing of that command to the display). Then, the path, keyboard and prompt commands are executed unseen, until echo is reactivated by executing the ECHO command with a trailing message which is displayed on the screen, followed by the version (VER) of your MS-DOS. Note the repeated reference to the C: drive which allows the path to be correctly set even if the user logs onto a drive other than C:.

Do remember to reboot the system with the **Ctrl-Alt-Del** key combination in order to activate the **autoexec.bat** file after you have changed it.

INDEX

BERNARD BABANI BP264

A Concise Advanced User's Guide to MS-DOS

☐ If you are a PC user and are at ease with the routine usage of its PC/MS-DOS operating system, but are looking for ways to improve your system's efficiency and productivity, while learning something new, then this CONCISE ADVANCED USER'S GUIDE TO MS-DOS will help you to do just that, in the shortest and most effective way.

☐ This book was written with the busy person in mind and, as such, it has an underlying structure based on "what you need to know first, appears first". Nonetheless, the book has also been designed to be circular, which means that you don't have to start at the beginning and go to the end.

☐ The book explains:

How to write both simple and advanced customised batch files which allow you to display what you want, and in the form and order you want it.

How the **ANSI.SYS** display and keyboard commands can be used to position the cursor on any part of the screen, change the intensity of the displayed characters or change their colour, or redefine keyboard keys so that by pressing such a key a complete command can be issued as if it was typed at the keyboard.

How the **edlin** line editor can be used to enter ESCape (ANSI.SYS) commands into a file so that simple menus can be built.

How the **DEBUG** program can be used to create, see and change the contents of any file, including those of programs written in assembler code.

How to find your way around the names and tasks of the CPU registers and the meaning of some simple assembler mnemonics.

How to write programs in assembly code, using **DEBUG**, which can control your screen and keyboard.

How to design and set up an interactive professional looking menu screen so that you or others can run program applications or packages easily.

☐ The book is relevant to both the PC-DOS and MS-DOS flavours of DOS, as implemented by IBM and other manufacturers of "compatible" microcomputers. It covers all versions of 2.x, 3.x and 4.x.

£2.95

ISBN 0-85934-209-3

00295

9 780859 342094